# Mutation-based Test Data Generation for Simulink Models using Genetic Algorithm and Simulated Annealing

Le Thi My Hanh, Khuat Thanh Tung, Nguyen Thanh Binh
DATIC Laboratory
University of Science and Technology
Danang, Vietnam
Email: ltmhanh{at}dut.udn.vn

*Abstract*—**Software testing is costly, labor intensive, and time consuming. Modern testing requires faults to be discovered at the earliest possible stages to decrease the cost of fixing errors in software development process. Thus, high level models such as Simulink models have become the focus of much verification effort and research. Mutation testing is a powerful and effective testing technique in terms of process automation and faults detection. Test case generation for Simulink that achieves a high mutation score is complicated. In this paper, we propose the automated test data generation approach based on mutation testing for Simulink models by using Genetic Algorithm (GA) and Simulated Annealing (SA) in order to improve the quality of test data. The approach has been applied to some different case studies and the obtained results are very promising.**

***Keywords-Mutation testing; Test data generation; Genetic Algorithm; Simulated Annealing; Simulink***

## I. INTRODUCTION

Software testing is an expensive process. It typically consumes more than half of the total development budget [1] but it is an effective way to estimate the confidence of software. Complete testing is impossible due to the huge input spaces involved. Therefore, it is desirable to seek techniques that will achieve testing rigor at an acceptable cost [2].

Test data generation is one of the most tedious tasks in the software testing process. As system size grows, manual test data generation places a great strain on mental resources and budget. This problem becomes especially serious when developers want to achieve high confidence in the correctness of their developed systems. Automated test data generation is a way forward to solve this problem and to increase testing efficiency.

The modern aim of testing is to discover faults at the earliest possible stages because the cost of fixing an error increases with the time between its introduction and detection. Thus high-level models have become the focus of much modern-day verification effort and research. Matlab/Simulink is a standard formalism that is widely used for modeling and simulating high-level designs of control applications in embedded systems engineering. The simulation facilities allow such models to be executed and observed. This property of Simulink turns out to be an advantage for effective dynamic testing [2].

Generation of test data for Simulink models is the focus of many researches. Zhan [3] built a framework including search-based test data generation for structural testing and mutation testing in Simulink using Simulated Annealing. However, the nature of mutation used in this work is a very simple one; only values on signal lines are perturbed. Ghani *et al.* [4] extended the work of Zhan, they used Simulated Annealing and Genetic Algorithms to generate test data for Simulink models based on branch coverage. He *et al.* [5] have translated Simulink models to formal specifications and generated test cases based on bounded model checking technique and mutation testing. Formal methods are hard to be implemented and require more mathematical knowledge. Translating Simulink models to formal language is possible for small models. However, this becomes difficult and very costly, if the size of the models is increasing. Li and Kumar [6] have translated Simulink models to an Input/Output Extended Finite Automaton, then generated test data. Godboley *et al.* [7] proposed to generate C/C++ code for Simulink models and subsequently generate test data using branch coverage for that code. Oh *et al.* [8] introduced a messy-GA for transition coverage of Simulink models. Their approach is able to achieve statistically significantly better coverage when compared to both random search and to a commercial tool.

Whilst there is much evidence that automated test data generation techniques can effectively automate the testing process, there has been little work on applying them in the context of mutation testing. In [33], mutation testing was shown to subsume most structural criteria. Therefore, in our work, we propose a test data generation method for Simulink models based on mutation coverage using Genetic Algorithm and Simulated Annealing instead of utilizing the branch coverage criterion that presented in [4]. Genetic Algorithm [29] is inspired by Darwin's theory about evolution. Algorithm is started with a set of solutions called population. Solutions from one population are taken and used to form a new population.

This is motivated by a hope, that the new population will be better than the old one. Solutions which are selected to form new solutions are selected according to their fitness - the more suitable they are the more chances they have to reproduce. Simulated Annealing [32] is a stochastic relaxation technique, which has its origin in statistical mechanics. It is based on an analogy from the annealing process of solids, where a solid is heated to a high temperature and gradually cooled in order for it to crystallize in a low energy configuration. SA can be seen as one way of trying to allow the basic dynamics of hill-climbing to also be able to escape local optima of poor solution quality. By using the GA and SA, our approach can optimize the test suites which are randomly initialized to kill as many mutants as possible.

The rest of this paper is further organized as follows: Section 2 briefly introduces mutation testing, the Simulink environment and mutation testing for Simulink models. Section 3 shows several techniques for test data generation. In section 4, we propose the automated test data generation technique using Genetic Algorithm and Simulated Annealing. Experimental results are discussed in section 5. Section 6 presents the conclusion and future work.

## II. MUTATION TESTING FOR SIMULINK MODELS

### A. Mutation Testing

Mutation testing, a fault-based testing technique proposed by DeMillo *et al.* [9], focuses on measuring the quality of a test set according to its ability to detect specific faults. It works in the following way: a large number of simple faults, such as alterations to operators, constant values, data type and variables, are introduced into the program under test one at a time.

We use mutation operators to introduce systematic faults into original model. Mutation operators can be seen as representing common faults usually found in software. Thus, mutation operators are designed based on the experience of the target language usage and the most common faults. Mutation testing criterion was originally used for program test. In this paper, we use the set of mutation operators proposed in [10] to implement mutation testing for Simulink models.

When applying a mutation operator into model, *ie.* inserting a single fault into model, we obtain a faulty model, which is called a mutant. Then, test data should be generated to reveal the introduced fault. A test suite is considered good if it contains tests that are able to distinguish a large number of these mutants from the original design. If a mutant can be distinguished from the original model by at least one of the test cases in the test set, the mutant is considered to be killed. Otherwise the mutant is alive. Sometimes the mutants cannot be killed due to the semantic equivalence of the mutants and the original model. These mutants are called equivalent mutants. Worse still, determining whether a mutant is equivalent is undecidable [11], and so typically the decision is left for the tester to establish manually.

The proportion of mutants killed out of all non-equivalent mutants is called Mutation Score and is formally defined as follows:

$$Mutation\ Score = \frac{K}{T - E}$$

where $K$ is the number of mutants that has been killed, $T$ is the total number of mutants, and $E$ is the number of equivalent mutants.

The major disadvantages of mutation testing are the huge number of generated mutants, the cost of executing mutant models, test data generation in the wide range of input domains, and determining any equivalent mutants.

### B. Simulink

Simulink [12] is a block diagram environment for multi-domain simulation and model-based design. It supports simulation, automatic code generation, and continuous test and verification of embedded systems. Simulink provides a graphical editor, customizable block libraries, and solvers for modeling and simulating dynamic systems. It is integrated with Matlab, enabling to incorporate Matlab algorithms into models and export simulation results to Matlab for further analysis.

Simulink has been popularly used as a high-level system prototyping or design tool in many domains, including aerospace, automobile and electronics systems.

A Simulink model consists of two main elements: blocks and lines. Blocks are the functional units, used to generate, manipulate and output signals. Blocks are connected by lines that provide the mechanism to transfer signals across the connection. A block can be a parent container containing other blocks, each modeling a subsystem or sub-functionality. Blocks are taken from pre-defined block libraries (covering generic functions such as addition or logical operators, but also domains like fuzzy logic or network communication) and receive a specific number of input signals from which output signals are computed.

Simulink models consist of a set of blocks that are connected by signals specifying the flow of data. Stateful systems are modeled with the help of feedback loops. Models can be structured hierarchically with the help of subsystems, and can be simulated, analyzed, or compiled to code using the Matlab tool-suite and third-party products [5].

As a whole, a Simulink program receives a number of (potentially infinite) streams of input values via using Inport blocks and generates a number of output streams which are represented by Outport blocks.

Simulink plays an increasingly important role in system engineering, and the verification and validation of Simulink models are becoming vital to users [4]. Thus, automatic test data generation for Simulink model has an important meaning.

In our work, we perform test data generation for Simulink models that contain basic blocks in pre-defined libraries such as commonly used blocks, continuous, discrete, logic and relational operations, math operations, sinks and sources.

### C. Mutation Testing for Simulink models

Since being proposed, mutation testing has been applied for many programming languages such as Fortran, Ada, C, Java, C#, etc. In this study, we adopt mutation testing for the designs in Simulink.

TABLE I. MUTATION OPERATORS.

| Operator | Description |
|---|---|
| TRO | Type Replacement Operator |
| VNO | Variable Negation Operator |
| VCO | Variable Change Operator |
| CCO | Constant Change Operator |
| CRO | Constant Replacement Operator |
| SCO | Statement Change Operator |
| SSO | Statement Swap Operator |
| DCO | Delay Change Operator |
| ROR | Relational Operator Replacement Operator |
| AOR | Arithmetic Operator Replacement Operator |
| ASR | Arithmetic Sign Replacement Operator |
| LOR | Logical Operator Replacement Operator |
| BRO | Block Removal Operator |
| SRO | Subsystem Replacement Operator |

Table I presents the set of mutation operators which are proposed in our previous work [10]. In order to generate mutants, we use 11 mutation operators to introduce systematic faults into the original Simulink model. We ignore three mutation operators TRO, SRO, DCO because they generate many equivalent mutants and they are less effective for our case studies.

Mutants in Simulink models can be divided into two main groups. The first group includes the VCO and VNO operators that disturb signal in lines, whilst the second group contains the remaining operators which modify attributes of blocks.

Applying mutation testing for Simulink models consists of generating mutants, executing mutants and analyzing the results. If this process is manually done, it will require too much time. Thereby, we build MuSimulink tool to automate this process. The details of this tool were presented in [13].

Mutation testing suffers from the high computational cost of automated test case generation and execution, due to the large number of mutants that can be derived from the original program. Despite existing cost reduction techniques [14], the problem of the required computational time to execute tests against the original system and their mutants still remains. The costs of mutation testing mainly depend on the number of mutants generated and the number of test cases. In order to reduce the cost of executing test data generated for Simulink models, we propose the use of Parallel Computing Toolbox of Matlab to execute in parallel the mutant models with each test data generated locally on a multicore machine. The empirical results in [15] shown that the mutants execution time was significantly reduced. Thereby, we use this toolbox in order to execute the test data which are generated by using the GA and SA. In next section, we clarify several techniques that are often used for generating test set before representing our proposed approach.

## III. AUTOMATED TEST DATA GENERATION

Test data generation is one of the most important steps in software testing. Given an input specification and/or Inport blocks of Simulink model, a suitable set of test case needs to be generated to test model. This activity is usually conducted manually. However, manual test data generation is a hard, laborious and very time consuming activity. Automating this step in testing can greatly reduce effort and cost. There are many techniques for automation which are applied throughout the software development life cycle as well as software testing. In mutation testing, the problem of choosing which test cases to generate is directed by mutants. A set of test cases that can distinguish between the original model and its mutants is sensitive with respect to errors, and is thus thought to be a good sample of the usually infinitely large domain of test cases. There are some most prominent techniques of automated test data generation, including symbolic execution [34], model-based, combinatorial, adaptive random [35] and search-based testing [25]. The simulation facilities of Simulink allow such models to be executed and observed. It turns out to be an advantage for effective dynamic testing by using random or search-based test data generation approach. Therefore, in this paper, we consider two these methods to generate test data based on mutation testing.

### A. Random Test Data Generation

Random test-data generation is the cheapest automatic test generation approach [2]. However, it frequently fails to test all required features of the object under test, especially unusual features that are only exercised by a small fraction of the overall input domain.

Random approaches generate test input vectors with elements randomly chosen from appropriate domains. Input vectors are generated until some identified criterion has been satisfied such as the maximal number of test cases. Random testing may be an effective means of gaining an adequate test set for simple programs but may simply fail to generate appropriate data in any reasonable time-frame for more complex software [4].

For the majority of mutants, any random test will suffice (providing a use of random test generators), however a small percentage of hard-to-kill mutants require scrutinizing the model and hand-generating tests in order to identify the faults they contain. This is both time consuming and challenging.

### B. Search-based Test Data Generation

Search-based approaches have been developed to address a wide and diverse range of domains, including testing approaches based on interactions [16], agents [17], regression [18, 19], aspects [20], integration [21, 22], stress [23], mutation [24, 25] and web applications [26].

Search-based test data generation searches a test object's input domain to find test data automatically [27]. It uses fitness function in guiding search to achieve a target. The search-based

test data generation process includes two main activities, interpreting the test data generation problem as an optimization problem, and performing the optimization based search [25]. Converting a test data generation problem into an optimization problem is really about designing the strategy for evaluating the test data.

A wide range of different optimization and search techniques has been used. The most widely used methods are local search, ant colony [36], Tabu search [37], simulated annealing [32], and genetic algorithms [29]. However, whatever may be the search techniques employed, fitness function or cost function plays a major role to guide and seek input test data that achieve the highest mutation score. The dynamic test data search is not guaranteed to succeed. Clearly, a search will fail to find appropriate test data when no such data exists or input data specification of user is inappropriate.

It is common to use structural coverage and mutation coverage as adequacy criteria to generate test data [2]. Structural coverage is the coverage of a particular set of elements in the structure of the program and includes control-flow based and data-flow based criteria. Mutation coverage can be defined as causing each conjectured fault to be detected by at least one test. In [4], Ghani et al. applied search-based technique and structural coverage whilst we use mutation coverage to generate test data for Simulink models.

## IV. The Proposed Approach

The test case generation for mutation testing aims to find test cases that achieve the mutation score as high as possible. In this paper, we propose the use of the genetic algorithm and simulated annealing in order to determine the suitable test cases for a Simulink model, which kill its most mutants.

For mutation testing, it is unlikely that one test will kill all mutants, requiring that, each individual or solution must incorporate a sufficient number of tests to allow that individual to achieve a full mutation score - i.e. killing all mutants. This in itself is a hindrance as the number of tests required to kill all mutants is generally unknown a priori [28].

### A. Genetic Algorithms

Genetic algorithms are probabilistic search algorithms inspired from biology [29]. They represent an optimization tool where their aim is to find a solution to a given problem. Thus, genetic algorithms try to explore the problem search space in order to find progressively potential solutions through time. Based on inheritance, natural selection, mutation, and sexual reproduction, they try to give after many generations the optimal solution in a finite time.

Genetic algorithms operate by iteratively refining a set of solutions to an optimization problem through random changes and by combining features from existing solutions [30]. In this context the solutions are called individuals and the set of individuals is called the population. Each individual has a genome that represents its unique features in a standardized format. Common formats for genomes are bit-strings and arrays of real values.

#### 1) Individual Representation in GA

There is a lot of methods that are used to represent individuals such as binary string, permutation encoding, string of some values, tree of some objects, functions or commands [31]. The encoding scheme used is highly dependent on the problem to solve. In mutation testing, a test case cannot kill all mutants. Therefore, a good solution for mutation testing is a set of tests that kill the most mutants. In this paper, we consider an individual is a set of tests. For example, an individual can be expressed as $[a_1, \ldots, a_k]$, where $a_i$ is a test case for Simulink model. The environmental pressure is the set of mutants generated by all mutation operators.

#### 2) Selection technique in GA

In [29], Mitchell presented a number of different methods that can be used to select individuals for reproduction, such as Roulette Wheel Selection, Rank Selection, Boltzmann Selection. In our study, we use the Roulette Wheel Selection technique. With this method the parents are selected according to their fitness. Better individuals are having more chances to be selected as parents. It is the most common method for implementing fitness proportionate selection. Each individual is assigned a slice of circular Roulette wheel, and the size of slice is proportional to the individual fitness, that is, the bigger the value, the larger the size of slice is. The functioning of Roulette wheel algorithm is described as follows:

1. Compute the sum of all individuals fitness in population: s
2. Generate the random number between 1 and s.
3. Loop through the entire population and sum the fitness. When this sum is more than or equals to the value in step 2, stop and return this individual.

A problem with selecting individuals in this method is that there is a high probability of losing the highest fitness individual – it may not be selected at all, or it will be selected but crossover and mutation will reduce its fitness. Thus, we retain the best individual in the each generation before performing crossover and mutation.

#### 3) Crossover Operators in GA

It is the process in which genes are selected from the parent individuals and new offspring are created. Crossover can be performed with binary encoding, permutation encoding, value encoding and tree encoding [31]. There is some crossover techniques that are used for the GA such as single point crossover, multi-point crossover, uniform crossover, arithmetic crossover, value encoding crossover, etc. In automated test data generation, we care both single point crossover and two points crossover. In this recombination, we generate the random number in the range [0, 1]. If this value is below a user-defined threshold probability (the crossover rate), one or two random locus is chosen on an individual. In single point crossover, the gene segment from this point to end are swapped between parents. In two points crossover, the gene segment between two random generated points are swapped. For example, suppose the two parents are $[a_1, ...., a_m]$ and $[b_1, ..., b_m]$, the first point is $k$ ($1 \leq k \leq m$ - 1) and the second point is $n$ ($k + 1 \leq n \leq m$), so they would produce the offspring: $[a_1, ..., a_k, b_{k+1}, ..., b_n, a_{n+1}, ..., a_m]$ and $[b_1, ..., b_k, a_{k+1}, ..., a_n, b_{n+1}, ..., b_m]$.

*4) Mutation Operators in GA*

A common view in the GA community, dating back to Holland's book *Adaptation in Natural and Artificial Systems*, is that crossover is the major instrument of variation and innovation in GAs, with mutation insuring the population against permanent fixation at any particular locus and thus playing more of a background role. This differs from the traditional positions of other evolutionary computation methods, such as evolutionary programming and early versions of evolution strategies, in which random mutation is the only source of variation [29]. Crossover operation of genetic algorithms cannot generate quite different offspring from their parents because the acquired information is used to crossover the individual. An alternate operator, mutation, can search new areas in contrast to the crossover.

Mutation of an individual is strongly dependent on the choice of representation for an individual. In [31], Rahulet *et al.* represented some techniques that are used to create mutation for each individual such as binary encoding mutation, permutation encoding mutation, value encoding mutation, tree encoding mutation.

Every locus of individual has equal chance of being mutated, based on the mutation rate; if, for a given locus, a random number (in the range [0,1]) falls below the mutation probability, that locus will be mutated. In test data generation for mutation testing Simulink models, with each locus mutated, we will replace the test case in this locus by a random generated test case. For instance, assume the child individual is $[a_1, ..., a_m]$, and the randomly chosen locus is $k$ ($1 \leq k \leq m$), then test data $a_k$ would be replaced by another.

*5) Fitness Function for test data generation*

Fitness in our work is calculated based on the total number of killed mutants by all the tests combined into individual. If there are two individuals that have the same number of killed mutants, we will choose the individual that has the least useful test cases. A useful test case is the one that can kill at least one mutant which was not killed by previous test cases in the test set.

*6) The proposed GA for generating test data for Simulink models based on mutation testing*

Fig. 1 shows the proposed GA for generating mutation-based test data for Simulink models. The algorithm works by initializing the population with *numIndividual* individuals which are randomly generated. Each individual has *numTescase* test cases. Then, through each generation, the GA uses selection, crossover and mutation to refine the initial population.

**Input:** The size of population: *numIndividual*
Maximum number of generations: *numGeneration*
Number of test cases in each individual: *numTestCase*
The crossover rate: *crossRate*
The mutation rate: *mutationRate*
Crossover type: *typeOfCrossover*
**Output:** The best individual of population.
**Algorithm:**
$P_0$ = Initialize the population with *numIndividual* individuals which are randomly generated. Each individual has *numTestCase* test cases.

**for** *i* **from** 0 **to** *numGeneration* - 1
*ChildPopulation* = {}
Parallel executing test cases of each individual in $P_i$ and calculate the number of its killed mutants.
*bestFittest* = Get the best individual in $P_i$
Add *bestFittest* into *ChildPopulation*. (Elitism)
**while** (size(*ChildPopulation*) < *numIndividual*)
*parent1* = RouletteWheelSelection($P_i$)
*parent2* = RouletteWheelSelection($P_i$)
[*child1, child2*] = Crossover(*parent1, parent2, crossRate, typeOfCrossover*)
MutateIndividual(*child1, mutationRate*)
MutateIndividual(*child2, mutationRate*)
Add *child1, child2* into *ChildPopulation*.
**end while**
$P_{i+1}$ = *ChildPopulation*
**end for**
**return** GetBestFittest($P_i$)

Figure 1. Genetic Algorithm to generate test data for Simulink models.

In this GA, function RouletteWheelSelection(P) is used to select the individual for reproduction in population P by applying the Roulette Wheel Selection technique. Method Crossover performs the crossover of two parent individuals with crossover rate *crossRate* and type of crossover *typeOfCrossover* to create two new offsprings. Then, these offsprings are mutated by using function MutateIndividual with mutation rate *mutationRate*. After performing *numGeneration* generations, the algorithm will return the best individual of population which kills the most mutants by using method GetBestFittest.

*B. Simulated Annealing*

Simulated annealing [32] originated from the physical process of annealing, a process of cooling a material in a heat bath so that a minimal energy state is reached. Simulated annealing is a global optimization heuristic that is based on the local descent search strategy. It attempts to avoid getting stuck in a local optimum by accepting a controlled amount of inferior solutions, in the hope of finding a long-term reward – a global optimum. It controls the acceptance of inferior solutions by gradually changing the value of a parameter called temperature, from high to low. Any better solution is always accepted. A worse solution is accepted probabilistically based on the temperature. At high temperatures, virtually any solution is accepted. At very low temperatures, there is little chance of an inferior solution being accepted. At the end, the temperature is reduced to zero, this means that the program will not accept any inferior solutions. Thus, the search is reduced to a simple gradient ascent/descent search. Similar to the local search, the cost function and the definition of the neighborhood need to be defined specifically to the problem being addressed. For simulated annealing, there are some other generic factors that can affect the search process itself, such as the initial temperature and the choice of temperature reduction function.

*1) Cost Function in SA*

Let $f$ be the cost function. We recall that a test case cannot kill all mutants. We need to generate a test set to kill the most mutants. In this work, we use $f = 1 - Mutation\ Score$ to assess the quality of a test set. In theory of the SA algorithm, the new

solution will always be accepted if it has lower value of the cost function than the current solution. A new solution is good if it kills more mutants than the current solution in the context of mutation testing. This means that it has higher mutation score. Hence, we proposed the above cost function.

*2) The proposed SA for generating test data for Simulink models based on mutation testing*

Fig. 2 presents the proposed SA algorithm for the problem of searching test data for Simulink models based on mutation testing. It searches for a solution that minimizes the cost function *f*. The algorithm works by randomly selecting a new solution *newSolution* in the neighborhood of the current solutions *currentSolution* by replacing some test cases in *currentSolution* with new test cases which are randomly generated. Solutions with lower cost ($\delta \leq 0$) are always accepted. Moving to inferior solutions ($\delta > 0$) may be accepted: the probability of acceptance depends on the magnitude of $\delta$ and on a control parameter t called temperature. Starting from $t_0$, the temperature is gradually reduced during the search (according to cooling schedule $\alpha$) so as to progressively decrease the acceptance rate of inferior solutions. The search is stopped when the maximal number of iterations is reached.

**Input:** Temperature reduction function α
Initial temperature $t_0 > 0$.
Maximum number of iterations: *numIteration*
Number of test cases in the each solution: *numTestCase*
**Output:** *bestSolution*: the best test set.
**Algorithm:**
*bestSolution* = *currentSolution* = RandomGenerateTestSuite()
Parallel executing test cases in *bestSolution* and *currentSolution* to calculate its mutation score.
*i* = 0
**while** ( *i* < *numIteration*)
- Mutating some test cases in *currentSolution* by replacing them with other test cases that are randomly generated to create *newSolution*
- Parallel execution of test cases in *newSolution* to calculate its mutation score.
**if** ( *f (newSolution)* == 0 ) **then**
*bestSolution = newSolution*
**break**
**end if**
δ = *f (newSolution)* – *f (currentSolution)*
**if** (δ ≤ 0) **then**
*currentSolution = newSolution*
**else**
Generating random x in the range [0, 1].
**if** ($x < e^{-\delta/t}$ ) **then**
*currentSolution = newSolution*
**end if**
**end if**
**if** (Fitness (*currentSolution*) > Fitness (*bestSolution*)) **then**
*bestSolution = currentSolution*
**end if**
*t = t* * α
++*i*
**end while**
**return** *bestSolution*

Figure 2. Simulated Annealing to generate test data for Simulink models.

In this SA, function RandomGenerateTestSuite is used to generate test set randomly. Each test set has numTestCase test cases. Method Fitness(S) is used to calculate the fitness of test set S, which is the number of mutants of model killed by S. At the end of the SA, bestSolution will be returned as the best test set for the Simulink model.

## V. EXPERIMENTATION

We implemented both approaches in the MuSimulink tool and used four models from [4] and the Quadratic_v2 model from [3] in Table II to assess our approach.

TABLE II. EXPERIMENTAL MODELS.

| Model | No of Input Var | No of Mutants |
|---|---|---|
| SmplSw | 2 | 92 |
| Quadratic_v1 | 2 | 161 |
| RandMdl | 3 | 188 |
| Tiny | 3 | 144 |
| Quadratic_v2 | 3 | 140 |

The SmplSw model contains two Inport blocks receiving input data from external. It has also two Switch blocks, one Outport block, one Product block and two Sum blocks.

The Quadratic_v1 model represents a simple equation with two Inport blocks, three Outport blocks, two Sum blocks, two Constant blocks, three Product blocks and three Switch blocks.

The RandMdl model is more complex than the above two models. It comprises four Switch blocks which give a total of 16 combination paths. It has also three Inport blocks, one Outport block, three Sum blocks and three Product blocks.

The Tiny model has three inputs X, Y, Z. When the predicated expression:

$$((\mathrm{Y}-\mathrm{Z}) * (\mathrm{Z}-\mathrm{X}) \geq 1000 \; or \; (Z * Z) \geq 8950)$$

is true, its output will be (X + Y). In contrast, its output will be (Z * Z). The Quadratic_v2 model checks if an equation is quadratic or not. If it is quadratic, it determines whether this quadratic will have real or complex solutions.

The hardware platform used for the experiments is a PC CPU Intel Xeon E5520 2.27 GHz with 8 GB RAM.

As presented in section 4, the GA depends on several configuration parameters, such as population size (*numIndividual*), number of generations (*numGeneration*), value of crossover rate (*crossRate*) and mutation rate (*mutationRate*). The SA depends on the temperature reduction function ($\alpha$), the initial temperature (*T*), and the maximal number of iterations (*numIteration*). We have carried out experiments for several models with different parameters for the GA and SA. Table III shows the values that we configured for the GA. Table IV describes the parameter values used for the SA.

TABLE III. PARAMETERS OF THE GA.

| PARAMETER | Test Set 1 | Test Set 2 | Test Set 3 | Test Set 4 |
|---|---|---|---|---|
| *numIndividual* | 40 | 20 | 30 | 40 |
| *numGeneration* | 10 | 20 | 20 | 20 |
| *crossRate* | 0.9 | 0.9 | 0.9 | 0.9 |
| *mutationRate* | 0.5 | 0.5 | 0.5 | 0.5 |

TABLE IV. PARAMETERS OF THE SA.

| PARAMETER | Test Set 1 | Test Set 2 | Test Set 3 | Test Set 4 |
|---|---|---|---|---|
| $\alpha$ | 0.95 | 0.95 | 0.95 | 0.95 |
| $T$ | 50000 | 60000 | 90000 | 100000 |
| *numIteration* | 200 | 300 | 450 | 600 |

With GA, we use fixed values for the two parameters which are the crossover rate (0.9) and the mutation rate (0.5) because they give better mutation score of models than other values that we experimented. With SA, we choose a value of 0.95 for α and a value which is larger than 50000 for *T* in order to offset the probability of accepting a worse solution, aiming to converge toward a global optimum and avoid getting stuck at local optimum. These values were derived empirically from our experiments and were used for the case studies in this paper.

Below section reports the results on performing test data generation for the selected models using the random, GA and SA approaches. For each execution, we performed these methods with the same input data specification. Table V shows the results of the SmplSw model.

TABLE V. RESULTS OF SMPLSW MODEL.

| Test Set | Random | | GA | | SA | |
|---|---|---|---|---|---|---|
| | *No. of killed Mutants* | *Time (s)* | *No. of killed Mutants* | *Time (s)* | *No. of killed Mutants* | *Time (s)* |
| 1 | 79 | 10.5 | 86 | 4417.1 | 84 | 2143.7 |
| 2 | 77 | 10.9 | 87 | 4655.6 | 86 | 3350.8 |
| 3 | 80 | 10.7 | 89 | 7533.1 | 86 | 5043.8 |
| 4 | 59 | 14.1 | 86 | 9605.6 | 86 | 6777.7 |

Table VI presents the results of applying the random, GA and SA algorithms to generate input test data automatically for the Quadratic_v1 model.

Table VII, VIII and IX describe the results for the RandMdl, Tiny and Quadratic_v2 models, respectively.

TABLE VI. RESULTS OF QUADRATIC_V1 MODEL.

| Test Set | Random | | GA | | SA | |
|---|---|---|---|---|---|---|
| | *No. of killed Mutants* | *Time (s)* | *No. of killed Mutants* | *Time (s)* | *No. of killed Mutants* | *Time (s)* |
| 1 | 125 | 14.5 | 137 | 5842.4 | 139 | 2583.0 |
| 2 | 125 | 14.0 | 139 | 5551.5 | 139 | 3928.8 |
| 3 | 125 | 13.1 | 139 | 8760.8 | 139 | 5931.6 |
| 4 | 125 | 13.8 | 135 | 11679.2 | 139 | 8050.1 |

TABLE VII. RESULTS OF RANDMDL MODEL.

| Test Set | Random | | GA | | SA | |
|---|---|---|---|---|---|---|
| | *No. of killed Mutants* | *Time (s)* | *No. of killed Mutants* | *Time (s)* | *No. of killed Mutants* | *Time (s)* |
| 1 | 150 | 15.6 | 166 | 6279.9 | 163 | 3002.3 |
| 2 | 146 | 17.3 | 161 | 6380.8 | 163 | 4637.3 |
| 3 | 143 | 18.3 | 161 | 10015.9 | 166 | 7205.5 |
| 4 | 147 | 20.8 | 162 | 14167.2 | 165 | 9805.7 |

TABLE VIII. RESULTS OF TINY MODEL.

| Test Set | Random | | GA | | SA | |
|---|---|---|---|---|---|---|
| | *No. of killed Mutants* | *Time (s)* | *No. of killed Mutants* | *Time (s)* | *No. of killed Mutants* | *Time (s)* |
| 1 | 117 | 15.0 | 123 | 5509.2 | 122 | 2599.2 |
| 2 | 112 | 13.3 | 121 | 5946.1 | 122 | 3969.6 |
| 3 | 117 | 13.1 | 121 | 8797.4 | 125 | 5999.2 |
| 4 | 116 | 13.3 | 123 | 11910.3 | 125 | 8104.3 |

TABLE IX. RESULTS OF QUADRATIC_V2 MODEL.

| Test Set | Random | | GA | | SA | |
|---|---|---|---|---|---|---|
| | *No. of killed Mutants* | *Time (s)* | *No. of killed Mutants* | *Time (s)* | *No. of killed Mutants* | *Time (s)* |
| 1 | 101 | 26.6 | 108 | 11063.3 | 106 | 5380.0 |
| 2 | 93 | 24.0 | 108 | 9443.6 | 103 | 7884.5 |
| 3 | 94 | 24.9 | 108 | 16810.6 | 106 | 11694.1 |
| 4 | 101 | 25.2 | 108 | 21957.2 | 105 | 15855.6 |

Table X shows the average number of killed mutants for each model by the generated test data according to the random, GA and SA techniques. Additionally, Fig. 3 presents the graph illustrating more clearly the effectiveness of each test data generation approach.

TABLE X. THE AVERAGE RESULTS OF EXPERIMENTATIONS.

| Model | No. of Mutants | No. of killed Mutants | | |
|---|---|---|---|---|
| | | *Random* | *GA* | *SA* |
| SmplSw | 92 | 73.75 | 87 | 85.5 |
| Quadratic_v1 | 161 | 125 | 137.5 | 139 |
| RandMdl | 188 | 146.5 | 162.5 | 164.25 |
| Tiny | 144 | 115.5 | 122 | 123.5 |
| Quadratic_v2 | 140 | 97.25 | 108 | 105 |

Based on these experimental results on Simulink models, we found that the proposed solution has significantly improved the number of killed mutants when compared to the random test data generation. From table X, we can compute the average mutation score of five experimental models which is 77% for random, 85.7% for the GA and 85.5% for the SA. Actually, the proposed approaches kill 9% more mutants than the random test generation does on average. However, we can observe that the process of generating and optimizing test data is time consuming. This can be explained by the fact that none of the equivalent mutants was eliminated from the set of candidate mutants. Moreover, the GA and SA depend on configuration parameters and each model should have appropriate parameters

for it. In this experiment, though, we used the same values of parameters that are *crossRate, mutationRate, α* for all models.

With the SA, we can find that the third combination of parameters provides the highest mutation score in all of the models under test. With the GA, however, we cannot determine which combination of parameters among the four used combinations gives the highest number of killed mutants for all of the models. Each combination is only good for some models. Thus, many trials are necessary to investigate implementation choices and to study alternative combinations of parameter settings in order to detect the most appropriate combination.

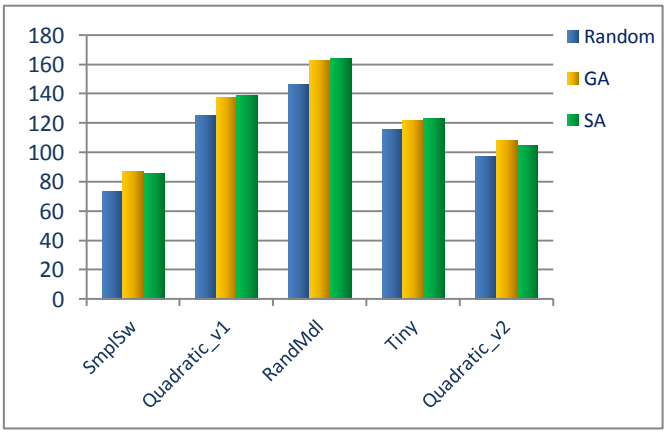


Figure 3. The average number of killed mutants of the case studies.

In general, the effectiveness of the algorithms depends on the specific characteristics of each model under test. The SA outperforms the GA in three models which are the Tiny, Quadratic_v1 and RandMdl whereas the GA is better than the SA in the remaining models. However, the SA is clearly better than the GA in terms of execution time.

The Quadratic_v2 and the Tiny models have many the hard-to-kill mutants, so the quality of the test cases increases very slowly and does not reach very high values. In this paper, the fitness calculations are performed based on the achieved mutation score. The drawback of this fitness is that it does not guide the search process by quantitatively measuring the closeness of killing specific mutants. In the mutation operator of the GA and in the each iterative step of the SA, we replace some test cases by others randomly and this process is not guided by the appropriate fitness function. Thereby, for a model that has many hard-to-kill mutants, its mutation score is not high. For these reasons, we intend to build a more effective fitness function in order to kill higher number of mutants in further work.

In our work, input data domain specification table plays an important role. The mutation score of model depends on this table. The better input domain description table, the higher number of killed mutants.

The mutant execution time is quite long and it depends on the number of iterations, so we have limited the number of iterations in the range of acceptable values. In order to increase execution performance, besides the proposed parallel execution, we are going to study to eliminate equivalent mutants. We tend also to refine fitness function by using mutation distance in later work.

## VI. Conclusion and Future Work

Automatic test data generation based on mutation coverage criterion is a time-consuming and complicated process. However, test data generated by this criterion is very effective for testing the safety-critical software such as designs in Simulink. In this work, we proposed the use of GA and SA approaches to generate mutation-based test inputs for Simulink models. The obtained results show that our approaches are more effective than random test generation in terms of the number of killed mutants. Beside, the parallel mutant execution approach reduced the test data generation time significantly. Although these results are not really high, they are very promising for us to continue our study in the future.

In future work, we intend to broaden the *MuSimulink* framework to include other search-based approaches such as artificial immune system and improve fitness function to increase effectiveness for the process of test data searching. We are also going to carry out more experiments to determine the influence of the GA and SA configuration parameters on the results as well as to automatically adjust parameters during the search process.

## References

[1] B. Beizer, Software Testing Techniques, 2nd ed.: Thomason Computer Press, 1990.

[2] Y. Zhan and J. Clark, "Automatic Test-Data Generation for Testing Simulink Model," University of York, Technical YCS-2004-382, 2004.

[3] Y. Zhan, "A Search-Based Framework for Automatic Test-Set Generation for Matlab/Simulink models," University of York, PhD Thesis, 2005.

[4] K. Ghani, J. A. Clark, and Y. Zhan, "Comparing Algorithms for Search-based Test Data Generation of Matlab Simulink Model," in 10th IEEE Congress on Evolutionary Computation (CEC '09), Trondheim, Norway, 2007.

[5] N. He, P. Rummer, and D. Kroening, "Test-Case Generation for Embedded Simulink via Formal Concept Analysis," in: DAC, San Diego, California, USA, 2011.

[6] M. Li and R. Kumar, "Model-Based Automatic Test Generation for Simulink/Stateflow using Extended Finite Automaton," 2011.

[7] S. Godboley, A. Sridhar, B. Kharpuse, D. P. Mohapatra, and B. Majhi, "Generation of Branch Coverage Test Data for Simulink/Stateflow Models using Crest Tool," International Journal of Advanced Computer Research, vol. III, no. 13, 2013, pp. 222-229.

[8] J. Oh, M. Harman, and S. Yoo, "Transition Coverage Testing for Simulink/Stateflow Models Using Messy Genetic Algorithms," in GECCO'11, Dublin, Ireland, 2011, pp. 1851-1858.

[9] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for Practicing for Programmer," IEEE computer, no. 11, 1978, pp. 34-41.

[10] Le Thi My Hanh and Nguyen Thanh Binh, "Mutation Operators for Simulink Models," in KSE 2012 - The fourth International Conference on Knowledge and Systems Engineering, Danang, 2012, pp. 54-59.

[11] T. A. Budd and D. Angluin, "Two notions of correctness and their relation," Acta Informatica, no. 18, 1982, pp. 31-45.

[12] The Matwork Inc, Simulink: http://www.mathworks.com/products/simulink/, 2014.

[13] L. T. M. Hanh and N. T. Binh, "Automatic Generation of Mutants for Simulink Models," in 16th National Conference: Selected Problems About It And Telecommunication, 2013, being published.

[14] P. R. Mateo and M. P. Usaola, "Mutation Testing Cost Reduction Techniques: A Survey," IEEE Software, vol. 27, no. 3, 2010, pp. 80-86.

[15] Le Thi My Hanh, Khuat Thanh Tung, Nguyen Thanh Binh, "Improving Mutation Execution in Mutation Testing for Simulink Models Using Parallel Computing," Journal of Science and Technology, University of Danang, vol. II, no. 1 (74), 2014, pp. 9-13.

[16] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge, "Constructing test suites for interaction testing," in Proc. of the 25th International Conference on Software Engineering (ICSE'03), 2003, pp. 38-48.

[17] C. Nguyen, A. Perini, P. Tonella, S. Miles, M. Harman, and M. Luck, "Evolutionary testing of autonomous software agents," in Proc. of the 8th International Conference on Autonomous Agents and Multiagent System (AAMAS'09), 2009, 521-528.

[18] K. R. Walcott, M. R. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time aware test suite prioritization," in Proc. of the International Symposium on Software Testing and Analysis (ISSTA'06), 2006, pp. 1-12.

[19] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in Proc. of the 2009 ACM International Conference on Software Testing and Analysis (ISSTA 09), 2009, pp. 201-212.

[20] M. Harman, F. Islam, T. Xie, and S. Wappler, "Automated test data generation for aspect-oriented programs," in Proc. of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09), 2009, pp. 185-196.

[21] T. E. Colanzi, W. K. G Assunsao, S.R. Vergilio, and A. T. R. Pozo, "Integration test of classes and aspects with a multi-evolutionary and coupling-based approach," in Proc. of the 3rd International Symposium on Search Based Software Engineering (SSBSE '11), 2011, Springer.

[22] L. C. Briand, J. Feng, and Y. Labiche, "Using genetic algorithms and coupling measures to devise optimal integration test orders," in Proc. of International Conference on Software Engineering and Knowledge Engineering (SEKE'02), 2002, pp. 43-50.

[23] C. D. Grosso, G. Antoniol, M. D. Penta, P. Galinier, and E. Merlo, "Improving network applications security: a new heuristic to generate stress testing data," in Proc. of the 2005 conference on Genetic and evolutionary computation, 2005, pp. 1037-1043.

[24] M. Harman, Y. Jia, and B. Langdon, "Strong higher order mutation based test data generation," in Proc. of the 8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11), 2011, pp. 212-222.

[25] Y. Zhan and J. A. Clark, "Search-based Mutation Testing for Simulink Models," ACM Digital Library, 2005,pp.1061-1068.

[26] N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," in Proc. of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11), 2011, pp. 3-12.

[27] R. Rajkumari and Dr. B. G. Geetha, "Automated Test Data Generation and Optimization Scheme Using Genetic Algorithm," in International Conference on Software and Computer Applications IPCSIT, Singapore, 2011, pp. 52-56.

[28] P. May, J. Timmis, and K. Mander, "Immune and Evolutionary Approaches to Software Mutation Testing," in International Conference on Artificial Immune System, Verlag Berlin Heidelberg, 2007, pp. 336-347.

[29] M. Mitchell, An Introduction to Genetic Algorithms, 5th ed.: The MIT Press, 1999.

[30] R. Nilsson, J. Offutt, and J. Mellin , "Test Case Generation for Mutation-based Testing of Timeliness," in Proceedings of the Second Workshop on Model Based Testing (MBT 2006), 2006, pp. 97-114.

[31] R. Malhotra, N. Singh, and Y. Singh, "Genetic Algorithms: Concepts, Design for Optimization of Process Controllers," Canadian Center of Science and Education, vol. 4, 2011, pp. 39-54.

[32] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," Science, no. 220(4598), 1983, pp. 671-680.

[33] A. J. Offutt and J. M. Voas, "Subsumption of Condition Coverage Techniques by Mutation Testing," Technical Report ISSE-TR-96-01, 1996.

[34] James C. King, "Symbolic execution and program testing," Communications of the ACM, 1976, pp. 385-394.

[35] P. Godefroid, N. Klarlund, and K. Sen, "DART : Directed Automated Random Testing," in Proc. of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05), 2005, pp. 213-223.

[36] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data generation via ant colony," in Proc. of the 9th annual conference on Genetic and evolutionary computation, London, England, 2007, pp. 1074-1081.

[37] Fred Glover, "Tabu search: A tutorial," Programming integer algorithm, heuristics, 1990, pp. 74-94.